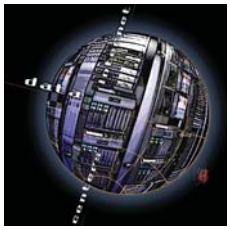
Recovery-Oriented Computing: Building Multitier Dependability



Building systems to recover fast may be more productive than aiming for systems that never fail. Because recovery is not immune to failure either, the authors advocate multiple lines of defense in managing failures.

George Candea
Stanford University

Aaron B. Brown
IBM T.J. Watson Research Center

Armando Fox
Stanford University

David Patterson
University of California, Berkeley

Personal computers often crash or freeze. Web sites become unavailable when we most need them. Upgrades can render systems unusable, and human operators regularly discombobulate the systems they administer. Clearly, software has not yet evolved to deal with our world. It is no surprise that system management dominates the total ownership costs of running large software infrastructures, with a large fraction of this cost going toward preventing and managing failures.

In a joint research effort of Stanford University and the University of California, Berkeley, the Recovery-Oriented Computing project studied techniques to help systems quickly recover from failures that are inevitable. ROC research was geared mainly toward Internet services because they present unique challenges: They can grow to immense proportions—Google has more than 100,000 compute nodes; they are subject to perpetual evolution, with weekly software updates being common; they face workloads that can vary by orders of magnitude over the course of a day; and, finally, within this dynamic environment, they are expected to run 24/7. We believe that most of what we learned from Internet services can also be applied to desktops, smaller network services, and other computing environments.

Our work builds on a vast body of computer dependability research. Daniel Siewiorek and Robert Swarz provide a standard text in this area, now in its third edition.[1] We approach dependability from a slightly different angle, however, to meet the constraints of rapidly evolving applications in the Internet service environment. ROC assumes that software will continue to be plagued by problems difficult to eradicate: elusive bugs, complex architectures, human operators subject to slips and lapses, unpredictable workloads, and so on. Regarding failures as a fact of life, ROC focuses on building systems that recover fast when a fault does strike, rather than on trying to prevent failures.

We implemented two building blocks for recovery, microreboot and system-level undo, which have proven very effective in handling failure. We are also developing benchmarks for quantifying the impact of these effects on system dependability.

RECOVERY DESIGN SPACE

A common formula defines system availability in terms of the relationship between mean-time-to-fail and mean-time-to-recover:

$$\text{Availability} = \text{MTTF} / (\text{MTTF} + \text{MTTR})$$

Given the progress made in increasing MTTF over the past decades, we argue that today it would be easier to approach 100 percent availability if we tried to reduce MTTR close to zero (very short recovery times) rather than continue striving to take MTTF to infinity (hardly ever fail); that is,

$$\lim_{\text{MTTR}\to 0} (\text{availability}) = 100\%$$

Of course, recovery must be not only fast but also correct. Like any software, a recovery mechanism

Published by the IEEE Computer Society
0018-9162/04/$20.00 © 2004 IEEE

cannot be flawless. Recovery-oriented systems must therefore defend themselves in depth, with multiple layers of recovery that can back each other up. We believe the design space for recovery spans two principal axes: cost of recovery and breadth (percentage) of recoverable failures.

Figure 1 illustrates this design space; we cast our work as an exploration of it.

The ROC project spanned several areas, ranging from fault detection and recovery to a survey of real-world, deployed system failures. Over the course of three years, we learned four general lessons that will help guide our design of future systems.

Fast recovery requires safeguarding system state. Recovering a failed system means bringing it back to the point at which it can serve its function at least as well as it did before failing. The system's *state*—the union of all the data the program uses to guide its operation, whether contained in a separate database on a remote server, in a file system, or inside its in-memory data structures—ultimately determines its ability to serve requests. To a great extent the cost of recovery is determined by how long it takes to reconstruct the system's state after failure and how much state gets lost before or during recovery.

For example, a simple Web server is a largely "stateless" system that serves data from a mostly read-only file system. Individual HTTP requests do not require the server to maintain state on their behalf, so the server can be rebooted safely. Subsequent HTTP requests will not know the server was restarted.

On the other hand, an operating system reads and modifies a variety of state on behalf of the applications it runs. Discarding this state would render the entire system useless. Accordingly, an OS is a very "stateful" program. Recovering from a problem such as a corrupt Windows registry could require reinstalling the entire operating system and applications, in effect recreating a large amount of state, which takes from hours to many days.

Most dependable systems rely on some form of rollback recovery,[2] using state checkpoints or activity logs to restore system state after a failure. Because rollback can be expensive, the key to fast recovery is protecting this state from corruption by safeguarding as much of it as possible from the program logic.

Overall dependability increases with the level of separation of data management from application logic, as long as programs access the data through well-defined, high-level interfaces. The inventors of databases noticed this property decades ago and leveraged it for persistent state.

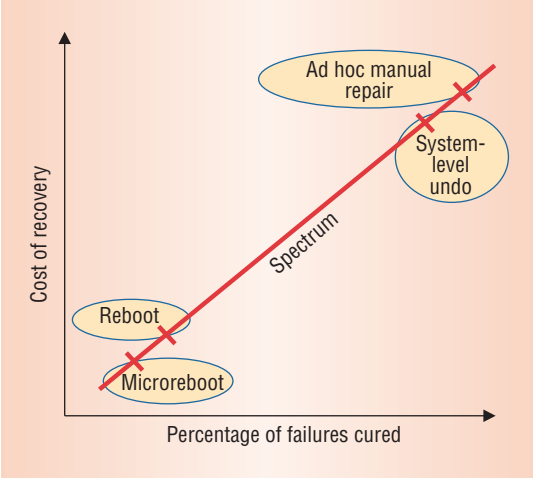


Figure 1. Software recovery design space. One axis captures the amount of breadth a technique can achieve in terms of failures it cures; the other axis refers to the general cost of employing that technique. In a sense, these axes capture a cost-benefit ratio for recovery.

Fine-grained workloads speed recovery. Another aspect influencing system recovery is the workload it serves. If the system designer can break the workload into small units that are independent of each other, then recovering from failure requires little state reconstruction, thus making recovery fast.

When rebooting a simple Web server, the only lost state is the HTTP requests that were in-flight at the time of the reboot. HTTP's stateless nature, combined with the client's ability to retry, preserves correctness across short-lived failures.

Implementing recovery in the platform is cost-effective. The two ROC prototypes described here implement recovery in the underlying platform rather than the applications. Although it is somewhat more difficult, this approach can leverage developed and tested recovery code across multiple applications, both present and forthcoming.

You can't improve what you can't measure. An important goal throughout the ROC project has been to develop suitable ways to benchmark the dependability of our systems, including the people who operate the systems and those who use them. The process of defining such benchmarks taught us where the systems' vulnerable points reside, and using the benchmarks helped guide our designs.

An added benefit of our benchmarking efforts was the ability to make a more persuasive, quantitative case for the recovery techniques and paradigms we developed, thus enhancing the prospects of their adoption in real systems.

WHEN IN DOUBT, MICROREBOOT

A first-line-of-defense recovery mechanism should be low cost and low overhead, with a good probability of repairing the problem, but a low opportunity cost in the event it doesn't work.

In industry, rebooting is generally accepted as a universal form of recovery for many software failures, even when the exact causes of failure are unknown. Rebooting provides a high-confidence way to reclaim stale or leaked resources. System administrators can attempt a reboot whether or not

Microreboots are largely as effective as full reboots, but 30 times faster.

the target system's software is responding. Rebooting is easy to implement and automate, and it returns software to the start state, which is often its best-understood and best-tested state. We therefore saw crash-restarting as a promising candidate for first-line recovery.

Microreboot: A technique for cheap recovery

Unfortunately, recovering from an unexpected crash can take a long time if extensive state reconstruction is required. Moreover, in systems that are not crash-safe, unexpected reboots can cause data loss.

A *microreboot* is the selective crash-restart of only those parts of a system that trigger the observed failure.[3] This technique aims to preserve the recovery advantages of rebooting while mitigating the drawbacks. In general, a small subset of components is often responsible for a global system failure, thus making the microreboot an effective technique for system-global recovery.

By localizing recovery to a small subset of components, microrebooting minimizes the amount of state loss and reconstruction. To further reduce state loss, we segregated the state that must survive the microreboot into separate state repositories that are themselves crash-safe. This separates the problem of data recovery from application-logic recovery and lets us perform the latter at finer grain than the process level.

We prototyped this approach in JBoss, a popular open-source application server written in Java. JBoss supports Java 2 Enterprise Edition, which allowed us to exploit J2EE's component-based programming framework. The JBoss modifications allowed selective microrebooting of small groups of Enterprise JavaBeans, the RPC-like handlers that make up a J2EE application.

Not surprisingly, microrebooting turned out to be one to two orders of magnitude faster than a full process restart: Microrebooting an EJB takes less than 600 milliseconds, whereas an application server process restart takes almost 20 seconds.[3] A microreboot also loses much less user work than a process restart.

The J2EE model stores persistent data in a relational database. We further modified JBoss so that applications store their session state—that is, state that must persist between a user's login and logout, but is not needed after the session ends—in a dedicated crash-safe state repository that is optimized for fast recovery. Session state typically lasts tens of minutes; most application servers store it in memory, so the session is lost if the server process crashes. Externalizing session state this way caused throughput to drop by less than 2 percent, and the increase in average request latency was not perceptible to humans.

Microreboots are largely as effective as full reboots but 30 times faster. In our prototype, microreboots recover from a large category of failures for which system administrators normally restart the application, including deadlocked or hung threads, memory leaks, and corrupt volatile data.

If a component microreboot doesn't correct the failure, we can progressively restart larger subsets of components. This is like navigating upward on Figure 1's spectrum to find the most advantageous cost-benefit ratio.

Lessons learned from microrebooting

Our prototype implementation yielded several lessons regarding its use in a recovery strategy.

Componentized software is synergistic with microreboots. In our prototype, we took advantage of J2EE's component framework and added a few new features.

Because the component-level reboot time is determined by how long the system takes to restart the component and the component takes to reinitialize, a microrebootable application should aim for components that are as small as possible. In the case of J2EE, applications are composed of EJBs that communicate through fairly narrow application programming interfaces (APIs), allowing the technique to exploit EJB boundaries as natural fine-grained recovery boundaries.

To tolerate a component microreboot gracefully, a modularized system must have loosely coupled components with well-defined, enforced boundaries. Direct references, such as pointers, cannot span these boundaries; the system can use either a state repository or the application platform to maintain indirect, microreboot-safe references outside the components.

This design approach shifts the burden of data management from the often-inexperienced application writers to the specialists who develop state repositories. This leads to *crash-only software* (http:// crash.stanford.edu)—software that is both crash-safe and fast-rebooting.

Fine-grained recovery requires accurate fault localization. Unlike whole-process restarts, fine-grained rebooting requires knowledge of which components are faulty, so the system must identify the location of faults more accurately.

While developing detection and localization tools for rapidly evolving systems may seem difficult, our

colleagues built an application-generic fault detection and localization program. Pinpoint (http://pinpoint.stanford.edu) uses statistical learning techniques to detect and localize likely application-level failures in component-based Internet services.

Assuming that most of the system is working most of the time, Pinpoint learns a baseline model of system behavior. During system operation, it looks for anomalies (relative to this model) in low-level behaviors that are likely to reflect high-level application faults, and it correlates these anomalies to their potential causes (software components) within the system. While Pinpoint does exhibit occasional false positives, the integration of Pinpoint and microreboots offers higher availability than recovery based on full process restart.

Nonintrusive recovery improves failure management. Microrebooting just the necessary components reduces not only recovery time but also its effects on the system's end users.

Figure 2 illustrates this point, showing comparative perceived availability. For each point t along the horizontal axis, a solid vertical bar indicates that, at time t, no end user perceived the service as unavailable. A gap in an interval $[t_1, t_2]$ indicates that some request, whose processing spanned $[t_1, t_2]$ in time, eventually failed, suggesting the site was down. When recovering with a microreboot, the user population is largely insulated from the effects of recovery, with close to no visible global downtime.

In a cluster-based system, the cost of failing over to a good node in the face of failure is nontrivial, as it often involves state transfer, changes in load dynamics, and so on. Cheap recovery can simplify failure management. In some configurations, we found it more effective to first microreboot the failed node, and then initiate failover only if the microreboot did not cure the failure. The benefits of not failing over outweighed the brief functionality disruption of a microreboot.[3]

A software tonic. When the cost of making a mistake in failure detection is negligible, automated recovery can use microreboots regardless of what the failure is. Even if the microreboot does not recover the system, but some other subsequent recovery mechanism does, the microreboot attempt adds only negligible time to the recovery process. For instance, in our system, microrebooting with up to a 98 percent false positive rate maintained higher availability than full restarts with no false positives.

Microrebooting is not a cure-all. Component-level crash-restart works best on transient hardware and software failures triggered by so-called "Heisenbugs." It is also effective against resource leaks and corruption of volatile data structures.

These bug classes are important and difficult to prevent with today's quality assurance processes, but do not represent all system failures. For some failure types, such as reproducible software bugs and corruption of persistent data, rebooting may not provide a fix; for others, such as misconfigurations or botched upgrades, it is never a fix.

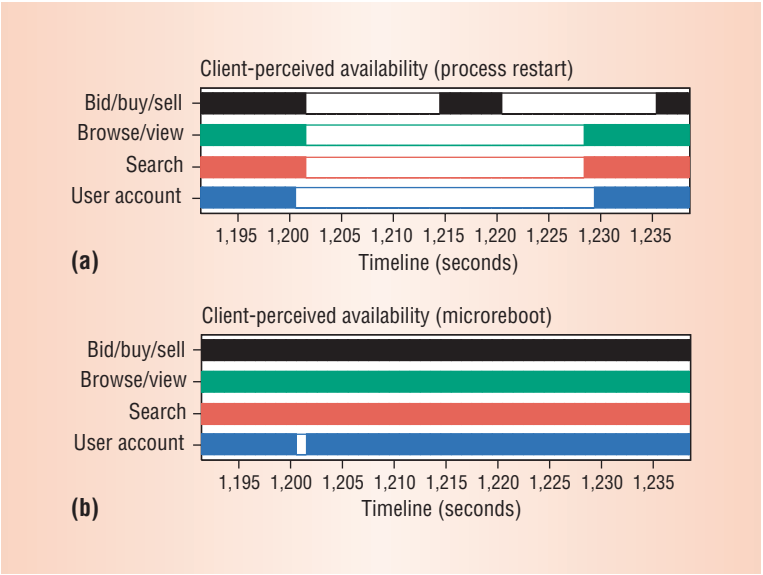


Figure 2. Service functionality availability. The graphs illustrate end-user-perceived availability of an online auction service's four functional groups during automated recovery from a user account component failure with (a) process restart and (b) microreboot.

THE BACKUP PLAN: UNDO/REDO

Software is not the only culprit in bringing large-scale software down; human errors also contribute to a significant fraction of downtime. These problems include incorrectly performed upgrades, configuration changes that unintentionally disable or degrade service, inopportune component shutdowns, and accidentally deleted data.

Undo/redo prototype

To extend recovery to all these failure types, we added a second line of defense based on the undo/redo pattern. Much like the undo feature in a word processor, our *system-level undo*[4] provides a more comprehensive and broadly applicable method for recovering from state-corrupting failure and human-operator error. Undo covers the operating system as well as user applications, so it incurs higher overhead—in the sense illustrated in Figure 1. It applies in situations that rebooting cannot fix and the costs—for example, of not recovering data—can be significantly higher than the possibly high costs of undo.

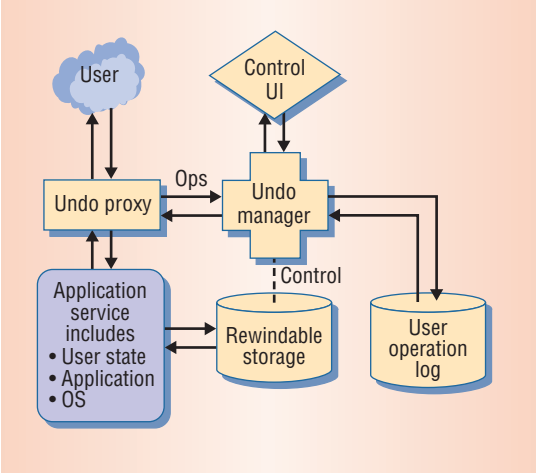


Figure 3. Undo/redo system design. The system wraps the application service, and the rewindable storage provides undo; the proxy logs users' requests and can replay them as part of redo. The undo manager coordinates the system timeline.

Figure 3 shows our prototype system. It wraps an e-mail server with an undo/redo layer that uses a proxy to log all external interactions with the server, such as e-mail delivery and mailbox manipulation. A rewindable storage layer offers undo functionality that can quickly restore a prior snapshot of all system state, including configuration state and user data as well as OS and application binaries. The redo functionality replays the logged external interactions via the proxy, thereby restoring end-user work lost during the undo operation while respecting the undo operation's system-level repairs.

For example, an operator who misconfigured a global spam filter to drop legitimate mail could use undo to revert the configuration change, then use redo to replay the lost e-mail traffic, thus restoring the dropped messages to their rightful mailboxes.

Because external interactions are likely to have different behavior when reexecuted after the undo operation, the undo/redo system provides a framework for detecting and handling such *paradoxes* via operator-supplied and application-specific consistency and compensation models.

Our benchmark experiments to assess the effectiveness of the undo/redo recovery mechanism have demonstrated its recovery power for state-corrupting human errors like failed software upgrades and misconfigurations. The changes added a small amount of runtime overhead in our prototype—several hundred milliseconds extra latency on some data-intensive e-mail commands. This did not affect throughput below the e-mail server's saturation point but reduced the saturation point by about 17 percent. We believe that minor architectural changes to our prototype will eliminate much of this overhead.

Lessons learned from undo/redo

While system-level undo is a more expensive level of defense than microrebooting, it can recover from a broader range of failures, including those induced by human operators.

Assimilating new recovery paradigms is hard. The greatest challenge in constructing and deploying system-level undo turned out not to be technical, but pedagogical. The rich undo/redo model that our prototype implementation supports requires a more sophisticated mental model than the simple undo found in productivity applications. Explaining this model to skeptical prospective system users—and indeed convincing ourselves that it could handle all paradox cases and could be trusted—proved an unexpected challenge.

Following in the footsteps of early transaction systems, we introduced the *spheres of undo* structuring concept. These spheres are "bubbles" of space and time that encapsulate undoable state, help visualize the temporal relationships between external observers and the state subject to undo/redo, and provide a mechanism for identifying all potential paradox situations.

The spheres proved to be a powerful tool for visualizing and reasoning about complex undo models, explaining them to potential users and suggesting undo/redo recovery extensions to other classes of systems, such as desktop computers and distributed server environments.

Identifying suitable target systems. A key question we faced when developing system-level undo/redo was how broadly we could apply it; this boils down to understanding the space of applications that can tolerate paradoxes. In addition to e-mail, we expect this approach to work well for other user-facing applications such as auctions, shopping, group calendar, even online banking. Many of these applications grew out of manual, human-driven processes and thus already include integral mechanisms for coping with the inherent inconsistency of complex real-world interactions.

Only a limited set of services, like block-level storage or embedded real-time control, are not good targets for undo/redo recovery because they either target other computers as end users or create immediate, irreversible changes to the external environment. The former cannot tolerate paradoxes without an extended model that supports distributed undo via multiple interacting spheres. In the latter case, compensation for paradoxes is unrealistic or impractically expensive.

Black-box recovery is expensive, but often worth it. We initially decided to treat the application service being recovered as a black box so that we could use undo/redo to recover from failed upgrades of the e-mail server itself and from misconfigurations of external tools like spam filters. We split the undo/redo system into two parts, an application-

independent core and a set of application-specific plug-ins to handle the e-mail proxying tasks and protocol knowledge.

Leveraging data and code already present in the e-mail server implementation would have greatly reduced the overall implementation complexity, as well as significantly reduced overhead. Alternatively, if the e-mail server had provided the needed hooks, we could have kept the black-box architecture while still reducing implementation complexity.

We believe that a fruitful direction of future research in undo/redo-based recovery will be to investigate the tradeoffs between recovery power and overhead at different points along the spectrum from our black-box approach to complete integration of undo/redo recovery within the application service itself.

SUITABLE DEPENDABILITY BENCHMARKS

A dependability benchmark consists of a system specification, a faultload, a workload, and a metric. When we found that no standard faultload existed for benchmarking the dependability of Internet services, we developed one through discussion with industrial experts.

Workloads, like faultloads, are also domain-specific. An e-shopping workload is different from an online auction workload, since only certain sequences of possible interactions with a particular service are meaningful; for example, a user must put an item in the shopping cart before paying for it. We modeled human end users using a Markov chain with states corresponding to end-user operations, similar to the approach used by Emmanuel Cecchet and coauthors.[5]

What metric is appropriate for an Internet service dependability benchmark?

Performability[6] is a good metric because it captures both service performance and availability. Unfortunately, it does not accurately distinguish between slow operations that involve many server resources, such as purchasing an item, and fast operations like accessing a static homepage. If the part of the system that handles heavyweight operations fails, the throughput of successful operations—and therefore the performability—can actually increase. This occurs because heavyweight operations fail immediately, rather than taking a long time to complete successfully, thus they allow numerous lightweight but "less useful" operations to complete instead.

Furthermore, performability does not capture the fact that user interactions are actually sequences of correlated operations that must succeed atomically to satisfy the user. Filling out payment and shipping forms is useless if the final "place order" operation fails.

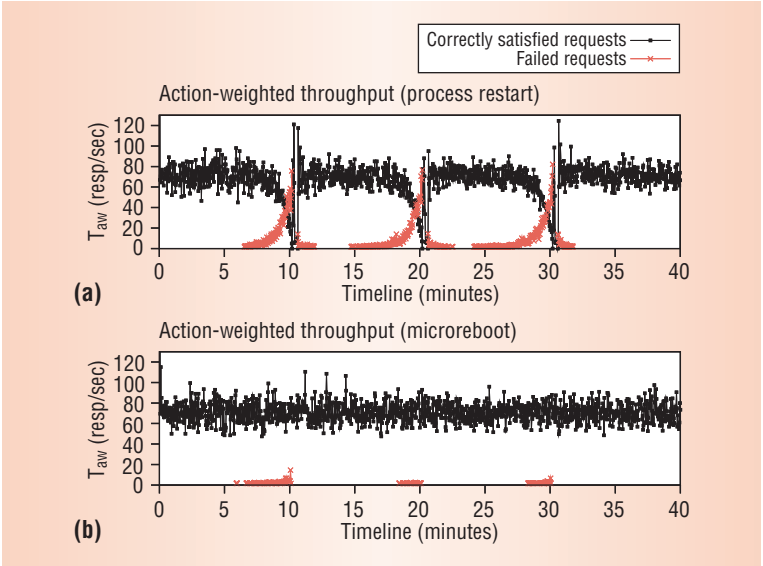


Figure 4. Action-weighted throughput measures end-user effects for (a) full restart and (b) component-level microreboot recoveries from three injected faults.

We evaluated availability using *action-weighted throughput*.[3] While similar to performability, this metric accounts for typical user interaction with a Web-based service as well as the different weights of different operations.

We assume that a user *session* begins with a login operation and ends with an explicit logout or abandonment of the site. Each session consists of a sequence of *user actions*; each user action is a sequence of *operations*; each operation in an action must succeed for that user action to be considered successful. When an operation fails, the entire user action fails. When an operation succeeds, it is counted as successful, unless it belongs to a user action that subsequently fails.

Figure 4 shows results for an evaluation of an online auction site using action-weighted throughput. In this case, a user action might take the form, "Search for a green SF Giants jersey and place a $20 bid on it." Individual user operations are the HTTP requests needed to complete this action.

We injected a sequence of three faults into the system every 10 minutes. Microrebooting much more effectively keeps the number of successfully served requests up and failed requests down. The tail of failed requests extending to the left of each injection point is composed of requests that were retroactively counted as failed once their containing action failed. Overall, 11,752 requests failed when recovering with a process restart, shown in the top graph; 233 requests failed when recovering by microrebooting one or more EJBs, shown in the bottom graph.

Action-weighted throughput takes the end user into account, but still does not provide quantitative accounting for the system administrators. The

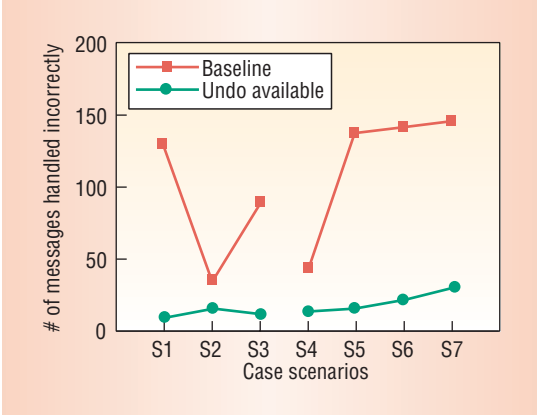


Figure 5. Benchmarking the human administrator component. For each of seven scenarios, the graph plots correctness and availability results with and without the undo recovery tool.

impact of human operator actions certainly translates into effects that are visible to the end user, but we needed a way to correlate the cause and observed effect, so we developed a new form of human-aware dependability benchmark[4] representing a cross between a traditional benchmark and a user study. While the human component makes this benchmark significantly more expensive than a traditional one, careful selection of the human participants and experimental protocol can minimize the extra cost.

We applied this human-aware benchmark to measure the effectiveness of our system-level undo prototype. We benchmarked the correctness and availability of the e-mail server with and without the undo recovery mechanism under two state-corrupting failure scenarios, using 12 graduate student subjects to perform recovery in seven case scenarios.

Figure 5 shows the subset of results demonstrating the dependability benefits (and costs) in these undo/redo recovery scenarios. In each case, the number of e-mail messages handled incorrectly decreased significantly with undo/redo. In some cases not shown here, undo-based recovery reduced availability compared to the baseline, due to its complex paradigm. This suggests we can further improve our prototype implementation.

Our work on microreboots was driven primarily by a desire to decrease the mean time to recover as a way to improve availability. In the background, we also wanted to improve recovery predictability. This has generated a new direction of research to improve the predictability of system behavior as a whole. We are developing a harness that uses component-level enforcement of invariants on resource usage, input, and output to maintain the system within the bounds of known good behavior (http://predictable.stanford.edu/).

We have also identified promising directions for continuing the undo/redo work. In addition to investigating the tradeoffs between black-box and white-box implementations, we are exploring the broader undo/redo application space. We are using the "spheres of undo" structuring concept to look at providing system-level undo in more complex systems—such as distributed services in which trust issues and multiway communication raise interesting complications with undo/redo and systems that contain hierarchies of configuration state, such as Windows-based desktop systems.

With the possible exception of human errors, noticing an error often takes much longer than diagnosing it, which in turn takes longer than a repair, especially if the repair is as low-cost as microreboot. We therefore see fault detection and diagnosis as an open challenge. We are starting a broader research project to systematically investigate the combination of statistical learning theory and simple fast recovery as a methodology for building large-scale self-recovering systems (http://rads.cs.berkeley.edu/). ■

Acknowledgments

The ROC project is comprised of the work of many graduate students at both Stanford and UC Berkeley, in addition to the authors of this article. We especially want to acknowledge the contributions of senior ROC students Mike Chen, James Cutler, Emre Kiciman, and David Oppenheimer, as well as Pete Broadwell, Andy Huang, and Ben Ling. More information about ROC is available at http://roc.cs.berkeley.edu/ and http://roc.stanford.edu/.

References

1. D.P. Siewiorek and R.S. Swarz, *Reliable Computer Systems: Design and Evaluation*, 3rd ed., AK Peters, 1998.
2. E.N.M. Elnozahy et al., "A Survey of Rollback-Recovery Protocols in Message-Passing Systems," *ACM Computing Surveys*, Sept. 2002, pp. 275-408.
3. G. Candea et al., "Microreboot—A Technique for Cheap Recovery," *Proc. 6th Symp. Operating Systems Design and Implementation* (OSDI), Usenix Assoc., 2004, pp. 31-44.
4. A.B. Brown and D.A. Patterson, "Undo for Operators: Building an Undoable E-mail Store," *Proc. Usenix Ann. Tech. Conf.*, Usenix Assoc., 2003, pp. 1-14.
5. E. Cecchet, J. Marguerite, and W. Zwaenepoel, "Performance and Scalability of EJB Applications," *Proc. 17th Conf. Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA 02), ACM Press, 2002, pp. 246-261.

6. J.F. Meyer, "On Evaluating the Performability of Degradable Computer Systems," *IEEE Trans. Computers*, Aug. 1980, pp. 720-731.

George Candea *is in the final year of his PhD program in computer science at Stanford University. His research interests center on large-scale software system dependability, including practical implementation of microreboots as a building block for high availability and the "crash-only software" design philosophy for applications that crash safely and recover fast. Candea received an SB and MEng in computer science from the Massachusetts Institute of Technology. Contact him through the Web at www.cs.stanford.edu/~candea.*

Aaron B. Brown *is a research staff member in the Adaptive Systems Department at IBM's T.J. Watson Research Center. He completed the work reported here while at the University of California, Berkeley. His research interests include quantifying and reducing IT management complexity, improving the dependability of business systems, and benchmarking nontraditional aspects of IT systems. Brown received a PhD in computer science from the University of California, Berkeley. Contact him at abbrown@us.ibm.com.*

Armando Fox *is an assistant professor of computer science at Stanford University. His research interests include mobile computing, Internet services, and system dependability. Fox received a PhD in computer science from the University of California, Berkeley. Contact him at fox@cs.stanford.edu.*

David Patterson *holds the Pardee Chair of Computer Science at the University of California, Berkeley. He led the design and implementation of the RISC architecture as well as the RAID storage project. Patterson received a PhD in electrical engineering from the University of California, Los Angeles. He is president of the ACM and a member of the National Academy of Engineering and the IEEE. Contact him at patterson@cs.berkeley.edu.*